Chapter 14 Working with the Screen, Printer, and Keyboard

14-1. Streams and C

Before you get to the details of program input/output, you need to learn about streams. All C input/output is done with streams, no matter where input is coming from or where output is going to. As you will see later, this standard way of handling all input and output has definite advantages for the programmer. Of course, this makes it essential that you understand what streams are and how they work. First, however, you need to know exactly what the terms *input* and *output* mean.

14-1-1. What Exactly Is Program Input/Output?

As you learned earlier in this book, a C program keeps data in random access memory (RAM) while executing. This data is in the form of variables, structures, and arrays that have been declared by the program. Where did this data come from, and what can the program do with it?

- Data can come from some location external to the program. Data moved from an external location into RAM, where the program can access it, is called *input*. The keyboard and disk files are the most common sources of program input.
- Data can also be sent to a location external to the program; this is called *output*. The most common destinations for output are the screen, a printer, and disk files.

Input sources and output destinations are collectively referred to as *devices*. The keyboard is a device, the screen is a device, and so on. Some devices (the keyboard) are for input only, others (the screen) are for output only, and still others (disk files) are for both input and output. This is illustrated in Figure 14.1.

Whatever the device, and whether it's performing input or output, C carries out all input and output operations by means of streams.

14-1-2. What Is a Stream?

A *stream* is a sequence of characters. More exactly, it is a sequence of bytes of data. A sequence of bytes flowing into a program is an input stream; a sequence of bytes flowing out of a program is an output stream. By focusing on streams, you don't have to worry as much about where they're going or where they originated. The major advantage of streams, therefore, is that input/output programming is *device independent*. Programmers don't need to write special input/output functions for each device (keyboard, disk, and so on). The program sees input/output as a continuous stream of bytes no matter where the input is coming from or going to.

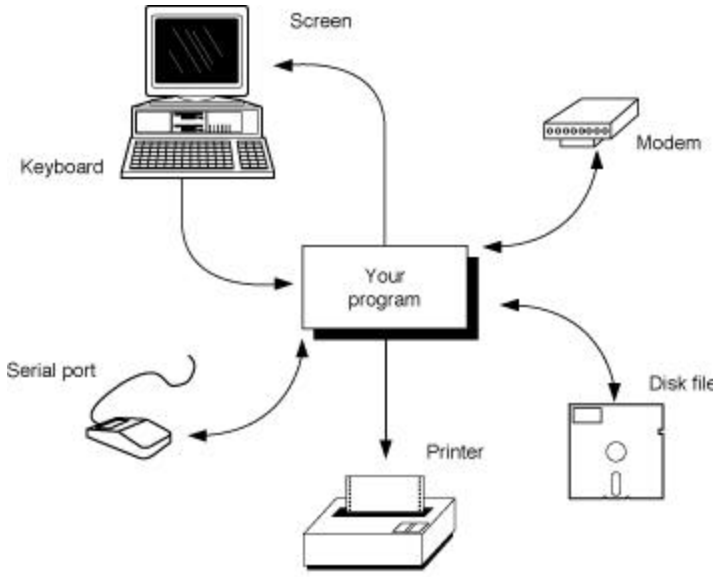

Figure 14-1. Input and output can take place between your program and a variety of external devices.

Every C stream is connected to a file. In this context, the term *file* doesn't refer to a disk file. Rather, it is an intermediate step between the stream that your program deals with and the actual physical device being used for input or output. For the most part, the beginning C programmer doesn't need to be concerned with these files, because the details of interactions between streams, files, and devices are taken care of automatically by the C library functions and the operating system.

14-1-3. Text Versus Binary Streams

C streams fall into two modes: text and binary. A *text* stream consists only of characters, such as text data being sent to the screen. Text streams are organized into lines, which can be up to 255 characters long and are terminated by an end-of-line, or newline, character. Certain characters in a text stream are recognized as having special meaning, such as the newline character. This chapter deals with text streams.

A *binary* stream can handle any sort of data, including, but not limited to, text data. Bytes of data in a binary stream aren't translated or interpreted in any special way; they are read and written exactly as-is. Binary streams are used primarily with disk files, which are covered on Day 16, "Using Disk Files."

14-1-4. Predefined Streams

ANSI C has three predefined streams, also referred to as the *standard input/output files*. If you're programming for an IBM-compatible PC running DOS, two additional standard streams are available to you. These streams are automatically opened when a C program starts executing and are closed when the program terminates. The programmer doesn't need to take any special action to make these streams available. Table 14.1 lists the standard streams and the devices they normally are connected with. All five of the standard streams are text-mode streams.

Table 14.1. The five standard streams.

Name	Stream	Device
stdin	Standard input	Keyboard
stdout	Standard output	Screen
stderr	Standard error	Screen
stdprn*	Standard printer	Printer (LPT1:)
stdaux*	Standard auxiliary	Serial port (COM1:)
*Supported only under DOS.		

Whenever you have used the printf() or puts() functions to display text on-screen, you have used the stdout stream. Likewise, when you use gets() or scanf() to read keyboard input, you use the stdin stream. The standard streams are opened automatically, but other streams, such as those used to manipulate information stored on disk, must be opened explicitly. You'll learn how to do this on Day 16. The remainder of this chapter deals with the standard streams.

14-2. C's Stream Functions

The C standard library has a variety of functions that deal with stream input and output. Most of these functions come in two varieties: one that always uses one of the standard streams, and one that requires the programmer to specify the stream. These functions are listed in Table 14.2. This table doesn't list all of C's input/output functions, nor are all of the functions in the table covered in this chapter.

Table 14.2. The standard library's stream input/output functions.

Uses One of the Standard Streams	Requires a Stream Name	Description
printf()	fprintf()	Formatted output
vprintf()	vfprintf()	Formatted output with a variable argument list
puts()	fputs()	String output

putchar()	putc(), fputc()	Character output
scanf()	fscanf()	Formatted input
gets()	fgets()	String input
getchar()	getc(), fgetc()	Character input
perror()		String output to stderr only

All these functions require that you include STDLIB.H. The function perror() may also require STDLIB.H. The functions vprintf() and vfprintf() also require STDARGS.H. On UNIX systems, vprintf() and vfprintf() may also require VARARGS.H. Your compiler's Library Reference will state whether any additional or alternative header files are needed.

14-2-1. An Example

The short program in Listing 14.1 demonstrates the equivalence of streams.

Listing 14.1. The equivalence of streams.

```
1:  /* Demonstrates the equivalence of stream input and output. */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     char buffer[256];
7:
8:     /* Input a line, then immediately output it. */
9:
10:    puts(gets(buffer));
11:
12:    return 0;
13: }
```

On line 10, the gets() function is used to input a line of text from the keyboard (stdin). Because gets() returns a pointer to the string, it can be used as the argument to puts(), which displays the string on-screen (stdout). When run, this program inputs a line of text from the user and then immediately displays the string on-screen.

DO take advantage of the standard input/output streams that C provides.

DON'T rename or change the standard streams unnecessarily.

DON'T try to use an input stream such as stdin for an output function such as fprintf().

14-3. Accepting Keyboard Input

Most C programs require some form of input from the keyboard (that is, from stdin). Input functions are divided into a hierarchy of three levels: character input, line input, and formatted input.

14-3-1. Character Input

The character input functions read input from a stream one character at a time. When called, each of these functions returns the next character in the stream, or EOF if the end of the file has been reached or an error has occurred. EOF is a symbolic constant defined in STDIO.H as -1. Character input functions differ in terms of buffering and echoing.

- Some character input functions are *buffered*. This means that the operating system holds all characters in a temporary storage space until you press Enter, and then the system sends the characters to the stdin stream. Others are *unbuffered,* meaning that each character is sent to stdin as soon as the key is pressed.
- Some input functions automatically *echo* each character to stdout as it is received. Others don't echo; the character is sent to stdin and not stdout. Because stdout is assigned to the screen, that's where input is echoed.

The uses of buffered, unbuffered, echoing, and nonechoing character input are explained in the following sections.

The getchar() Function

The function getchar() obtains the next character from the stream stdin. It provides buffered character input with echo, and its prototype is

```
int getchar(void);
```

The use of getchar() is demonstrated in Listing 14.2. Notice that the putchar() function, explained in detail later in this chapter, simply displays a single character on-screen.

Listing 14.2. The getchar() function.

```
1: /* Demonstrates the getchar() function. */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7:     int ch;
8:
9:     while ((ch = getchar()) != `\n')
10:         putchar(ch);
11:
12:    return 0;
13: }
This is what's typed in.
This is what's typed in.
```

ANALYSIS:] On line 9, the getchar() function is called and waits to receive a character from stdin. Because getchar() is a buffered input function, no characters are received until you press Enter. However, each key you press is echoed immediately on the screen.

When you press Enter, all the characters you entered, including the newline, are sent to stdin by the operating system. The getchar() function returns the characters one at a time, assigning each in turn to ch.

Each character is compared to the newline character \n and, if not equal, displayed on-screen with putchar(). When a newline is returned by getchar(), the while loop terminates.

The getchar() function can be used to input entire lines of text, as shown in Listing 14.3. However, other input functions are better suited for this task, as you'll learn later in this chapter.

Listing 14.3. Using the getchar() function to input an entire line of text.

```
1: /* Using getchar() to input strings. */
2:
3: #include <stdio.h>
4:
5: #define MAX 80
6:
7: main()
8: {
9:     char ch, buffer[MAX+1];
10:     int x = 0;
11:
12:     while ((ch = getchar()) != `\n' && x < MAX)
13:         buffer[x++] = ch;
14:
15:     buffer[x] = `\0';
16:
17:     printf("%s\n", buffer);
18:
19:     return 0;
20: }
This is a string
This is a string
```

ANALYSIS: This program is similar to Listing 14.2 in the way that it uses getchar(). An extra condition has been added to the loop. This time the while loop accepts characters from getchar() until either a newline character is reached or 80 characters are read. Each character is assigned to an array called buffer. When the characters have been input, line 15 puts a null on the end of the array so that the printf() function on line 17 can print the entered string.

On line 9, why was buffer declared with a size of MAX + 1 instead of just MAX? If you declare buffer with a size of MAX + 1, the string can be 80 characters plus a null terminator. Don't forget to include a place for the null terminator at the end of your strings.

The getch() Function

The getch() function obtains the next character from the stream stdin. It provides unbuffered character input without echo. The getch() function isn't part of the ANSI standard. This means that it might not be available on every system. Additionally, it might require that different header files be included. Generally, the prototype for getch() is in the header file CONIO.H, as follows:

```
int getch(void);
```

Because it is unbuffered, getch() returns each character as soon as the key is pressed, without waiting for the user to press Enter. Because getch() doesn't echo its input, the characters aren't displayed on-screen. Listing 14.4 illustrates the use of getch().

WARNING: The following listing uses getch(), which is not ANSI-compliant. You should be careful when using non-ANSI functions, because there is no guarantee that all compilers support them. If you get errors from the following listing, it might be because your compiler doesn't support getch().

Listing 14.4. Using the getch() function.

```
1: /* Demonstrates the getch() function. */
2: /* Non-ANSI code */
3: #include <stdio.h>
4: #include <conio.h>
5:
6: main()
7: {
8:     int ch;
9:
10:     while ((ch = getch()) != `\r')
11:         putchar(ch);
12:
13:     return 0;
14:}
Testing the getch() function
```

ANALYSIS: When this program runs, getch() returns each character as soon as you press a key--it doesn't wait for you to press Enter. There's no echo, so the only reason that each character is displayed on-screen is the call to putchar(). To get a better understanding of how getch() works, add a semicolon to the end of line 10 and remove line 11 (putchar(ch)). When you rerun the program, you will find that nothing you type is echoed to the screen. The getch() function gets the characters without echoing them to the screen. You know the characters are being gotten because the original listing used putchar() to display them.

Why does this program compare each character to \r instead of to \n? The code \r is the escape sequence for the carriage return character. When you press Enter, the keyboard device sends a carriage return to stdin. The buffered character input functions automatically translate the carriage return to a newline, so the program must test for \n to determine whether Enter has been pressed. The unbuffered character input functions don't translate, so a carriage return is input as \r, and that's what the program must test for.

Listing 14.5 uses getch() to input an entire line of text. Running this program clearly illustrates that getch() doesn't echo its input. With the exception of substituting getch() for getchar(), this program is virtually identical to Listing 14.3.

Listing 14.5. Using the getch() function to input an entire line.

```
1: /* Using getch() to input strings. */
2: /* Non-ANSI code */
3: #include <stdio.h>
4: #include <conio.h>
5:
6: #define MAX 80
7:
8: main()
9: {
10:     char ch, buffer[MAX+1];
11:     int x = 0;
```

```
12:
13:     while ((ch = getch()) != `\r' && x < MAX)
14:         buffer[x++] = ch;
15:
16:     buffer[x] = `\0';
17:
18:     printf("%s", buffer);
19:
20:     return 0;
21:}
Here's a string
Here's a string
```

WARNING: Remember that getch() isn't an ANSI-standard command. This means that your compiler (and other compilers) might or might not support it. getch() is supported by Symantec and Borland. Microsoft supports _getch(). If you have problems using this command, you should check your compiler and see whether it supports getch(). If you're concerned about portability, you should avoid non-ANSI functions.

The getche() Function

This is a short section, because getche() is exactly like getch(), except that it echoes each character to stdout. Modify the program in Listing 14.4 to use getche() instead of getch(). When the program runs, each key you press is displayed on-screen twice--once as echoed by getche(), and once as echoed by putchar().

WARNING: getche() is not an ANSI-standard command, but many C compilers support it.

The getc() and fgetc() Functions

The getc() and fgetc() character input functions don't automatically work with stdin. Instead, they let the program specify the input stream. They are used primarily to read characters from disk files. See Day 16 for more details.

DO understand the difference between echoed and nonechoed input.

DO understand the difference between buffered and unbuffered input.

DON'T use non-ANSI standard functions if portability is a concern.

"Ungetting" a Character with ungetc()

What does "ungetting" a character mean? An example should help you understand. Suppose that your program is reading characters from an input stream and can detect the end of input only by reading one character too many.

For example, you might be inputting digits only, so you know that input has ended when the first nondigit character is encountered. That first nondigit character might be an important part of subsequent data, but it has been removed from the input stream. Is it lost? No, it can be "ungotten" or returned to the input stream, where it is then the first character read by the next input operation on that stream.

To "unget" a character, you use the ungetc() library function. Its prototype is

```
int ungetc(int ch, FILE *fp);
```

The argument ch is the character to be returned. The argument *fp specifies the stream that the character is to be returned to, which can be any input stream. For now, simply specify stdin as the second argument: ungetc(ch, stdin);. The notation FILE *fp is used with streams associated with disk files; you'll learn about this on Day 16.

You can unget only a single character to a stream between reads, and you can't unget EOF at any time. The function ungetc() returns ch on success and EOF if the character can't be returned to the stream.

Line Input

The line input functions read a line from an input stream--they read all characters up to the next newline character. The standard library has two line input functions, gets() and fgets().

The gets() Function

You were introduced to the gets() function on Day 10, "Characters and Strings." This is a straightforward function, reading a line from stdin and storing it in a string. The function prototype is

```
char *gets(char *str);
```

You probably can interpret this prototype by yourself. gets() takes a pointer to type char as its argument and returns a pointer to type char. The gets() function reads characters from stdin until a newline (\n) or end-of-file is encountered; the newline is replaced with a null character, and the string is stored at the location indicated by str.

The return value is a pointer to the string (the same as str). If gets() encounters an error or reads end-of-file before any characters are input, a null pointer is returned.

Before calling gets(), you must allocate sufficient memory space to store the string, using the methods covered on Day 10. This function has no way of knowing whether space pointed to by ptr is allocated; the string is input and stored starting at ptr in either case. If the space hasn't been allocated, the string might overwrite other data and cause program errors.

Listings 10.5 and 10.6 use gets().

The fgets() Function

The fgets() library function is similar to gets() in that it reads a line of text from an input stream. It's more flexible, because it lets the programmer specify the specific input stream to use and the maximum number of characters to be input. The fgets() function is often used to input text from disk files, which is covered on Day 16. To use it for input from stdin, you specify stdin as the input stream. The prototype of fgets() is

```
char *fgets(char *str, int n, FILE *fp);
```

The last parameter, FILE *fp, is used to specify the input stream. For now, simply specify the standard input stream, stdin, as the stream argument.

The pointer str indicates where the input string is stored. The argument n specifies the maximum number of characters to be input. The fgets() function reads characters from the input stream until a newline or end-of-line is

encountered or n - 1 characters have been read. The newline is included in the string and terminated with a \0 before it is stored. The return values of fgets() are the same as described earlier for gets().

Strictly speaking, fgets() doesn't input a single line of text (if you define a line as a sequence of characters ending with a newline). It can read less than a full line if the line contains more than *n* -1 characters. When used with stdin, execution doesn't return from fgets() until you press Enter, but only the first *n*-1 characters are stored in the string. The newline is included in the string only if it falls within the first *n*-1 characters. Listing 14.6 demonstrates the fgets() function.

Listing 14.6. Using the fgets() function for keyboard input.

```
1:  /* Demonstrates the fgets() function. */
2:
3:  #include <stdio.h>
4:
5:  #define MAXLEN 10
6:
7:  main()
8:  {
9:     char buffer[MAXLEN];
10:
11:    puts("Enter text a line at a time; enter a blank to exit.");
12:
13:    while (1)
14:    {
15:         fgets(buffer, MAXLEN, stdin);
16:
17:         if (buffer[0] == `\n')
18:             break;
19:
20:         puts(buffer);
21:    }
22:    return 0;
23: }
Enter text a line at a time; enter a blank to exit.
Roses are red
Roses are
 red
Violets are blue
Violets a
re blue
Programming in C
Programmi
ng in C
Is for people like you!
Is for pe
ople like
 you!
```

Line 15 contains the fgets() function. When running the program, enter lines of length less than and greater than MAXLEN to see what happens. If a line greater than MAXLEN is entered, the first MAXLEN - 1 characters are read by the first call to fgets(); the remaining characters remain in the keyboard buffer and are read by the next call to fgets() or any other function that reads from stdin. The program exits when a blank line is entered (lines 17 and 18).

14-3-2. Formatted Input

The input functions covered up to this point have simply taken one or more characters from an input stream and put them somewhere in memory. No interpretation or formatting of the input has been done, and you still have no way to input numeric variables. For example, how would you input the value 12.86 from the keyboard and assign it to a type float variable? Enter the scanf() and fscanf() functions. You were introduced to scanf() on Day 7, "Fundamentals of Input and Output." This section explains its use in more detail.

These two functions are identical, except that scanf() always uses stdin, whereas the user can specify the input stream in fscanf(). This section covers scanf(); fscanf() generally is used with disk file input and is covered on Day 16.

The scanf() Function's Arguments

The scanf() function takes a variable number of arguments; it requires a minimum of two. The first argument is a format string that uses special characters to tell scanf() how to interpret the input. The second and additional arguments are the addresses of the variable(s) to which the input data is assigned. Here's an example:

```
scanf("%d", &x);
```

The first argument, "%d", is the format string. In this case, %d tells scanf() to look for one signed integer value. The second argument uses the address-of operator (&) to tell scanf() to assign the input value to the variable x. Now you can look at the format string details.

The scanf() format string can contain the following:

- Spaces and tabs, which are ignored (they can be used to make the format string more readable).
- Characters (but not %), which are matched against nonwhitespace characters in the input.
- One or more *conversion specifications,* which consist of the % character followed by special characters. Generally, the format string contains one conversion specification for each variable.

The only required part of the format string is the conversion specifications. Each conversion specification begins with the % character and contains optional and required components in a certain order. The scanf() function applies the conversion specifications in the format string, in order, to the input fields. An *input field* is a sequence of nonwhitespace characters that ends when the next white space is encountered or when the field width, if specified, is reached. The conversion specification components include the following:

- The optional assignment suppression flag (*) immediately follows the %. If present, this character tells scanf() to perform the conversion corresponding to the current conversion specifier but to ignore the result (not assign it to any variable).
- The next component, the field width, is also optional. The field width is a decimal number specifying the width, in characters, of the input field. In other words, the field width specifies how many characters from stdin scanf() should examine for the current conversion. If a field width isn't specified, the input field extends to the next white space.
- The next component is the optional precision modifier, a single character that can be h, l, or L. If present, the precision modifier changes the meaning of the type specifier that follows it. Details are given later in this chapter.
- The only required component of the conversion specifier (besides the %) is the type specifier. The type specifier is one or more characters that tell scanf() how to interpret the input. These characters are listed and described in Table 14.3. The Argument column lists the required type of the corresponding variable. For example, the type specifier d requires int * (a pointer to type int).

Table 14.3. The type specifier characters used in scanf() conversion specifiers.

Type	Argument	Meaning of Type
d	int *	A decimal integer.

i	int *	An integer in decimal, octal (with leading 0), or hexadecimal (with leading 0X or 0x) notation.
o	int *	An integer in octal notation with or without the leading 0.
u	unsigned int *	An unsigned decimal integer.
x	int *	A hexadecimal integer with or without the leading 0X or 0x.
c	char *	One or more characters are read and assigned sequentially to the memory location indicated by the argument. No terminating \0 is added. If a field width argument isn't given, one character is read. If a field width argument is given, that number of characters, including white space (if any), is read.
s	char *	A string of nonwhitespace characters is read into the specified memory location, and a terminating \0 is added.
e,f,g	float *	A floating-point number. Numbers can be input in decimal or scientific notation.
[...]	char *	A string. Only the characters listed between the brackets are accepted. Input ends as soon as a nonmatching character is encountered, the specified field width is reached, or Enter is pressed. To accept the] character, list it first:[]...]. A \0 is added at the end of the string.
[^...]	char *	The same as [...], except that only characters not listed between the brackets are accepted.
%	None	Literal %: Reads the % character. No assignment is made.

Before seeing some examples of scanf(), you need to understand the precision modifiers, which are listed in Table 14.4.

Table 14.4. The precision modifiers.

Precision Modifier	Meaning
H	When placed before the type specifier d, i, o, u, or x, the modifier h specifies that the argument is a pointer to type short instead of type int. On a PC, the type short is the same as type int, so the h precision modifier is never needed.
L	When placed before the type specifier d, i, o, u, or x, the modifier l specifies that the argument is a pointer to type long. When placed before the type specifier e, f, or g, the modifier l specifies that the argument is a pointer to type double.
L	When placed before the type specifier e, f, or g, the modifier L specifies that the argument is a pointer to type long double.

Handling Extra Characters

Input from scanf() is buffered; no characters are actually received from stdin until the user presses Enter. The entire line of characters then "arrives" from stdin, and is processed, in order, by scanf(). Execution returns from scanf() only when enough input has been received to match the specifications in the format string. Also, scanf() processes only enough characters from stdin to satisfy its format string. Extra, unneeded characters, if any, remain waiting in stdin. These characters can cause problems. Take a closer look at the operation of scanf() to see how.

When a call to scanf() is executed and the user has entered a single line, you can have three situations. For these examples, assume that scanf("%d %d", &x, &y); is being executed; in other words, scanf() is expecting two decimal integers. Here are the possibilities:

- The line the user inputs matches the format string. For example, suppose the user enters 12 14 followed by Enter. In this case, there are no problems. scanf() is satisfied, and no characters are left over in stdin.
- The line that the user inputs has too few elements to match the format string. For example, suppose the user enters 12 followed by Enter. In this case, scanf() continues to wait for the missing input. Once the input is received, execution continues, and no characters are left over in stdin.
- The line that the user enters has more elements than required by the format string. For example, suppose the user enters 12 14 16 followed by Enter. In this case, scanf() reads the 12 and the 14 and then returns. The extra characters, the 1 and the 6, are left waiting in stdin.

It is this third situation (specifically, those leftover characters) that can cause problems. They remain waiting for as long as your program is running, until the next time the program reads input from stdin. Then the leftover characters are the first ones read, ahead of any input the user makes at the time. It's clear how this could cause errors. For example, the following code asks the user to input an integer and then a string:

```
puts("Enter your age.");
scanf("%d", &age);
puts("Enter your first name.");
scanf("%s", name);
```

Say, for example, that in response to the first prompt, the user decides to be precise and enters 29.00 and then presses Enter. The first call to scanf() is looking for an integer, so it reads the characters 29 from stdin and assigns the value 29 to the variable age. The characters .00 are left waiting in stdin. The next call to scanf() is looking for a string. It goes to stdin for input and finds .00 waiting there. The result is that the string .00 is assigned to name.

How can you avoid this problem? If the people who use your programs never make mistakes when entering information, that's one solution--but it's rather impractical.

A better solution is to make sure there are no extra characters waiting in stdin before prompting the user for input. You can do this by calling gets(), which reads any remaining characters from stdin, up to and including the end of the line. Rather than calling gets() directly from the program, you can put it in a separate function with the descriptive name of clear_kb(). This function is shown in Listing 14.7.

Listing 14.7. Clearing stdin of extra characters to avoid errors.

```
1: /* Clearing stdin of extra characters. */
2:
3: #include <stdio.h>
4:
5: void clear_kb(void);
6:
7: main()
8: {
9:     int age;
10:     char name[20];
11:
12:     /* Prompt for user's age. */
13:
14:     puts("Enter your age.");
15:     scanf("%d", &age);
16:
17:     /* Clear stdin of any extra characters. */
```

```
18:
19:     clear_kb();
20:
21:     /* Now prompt for user's name. */
22:
23:     puts("Enter your first name.");
24:     scanf("%s", name);
25:     /* Display the data. */
26:
27:     printf("Your age is %d.\n", age);
28:     printf("Your name is %s.\n", name);
29:
30:     return 0;
31: }
32:
33: void clear_kb(void)
34:
35: /* Clears stdin of any waiting characters. */
36: {
37:     char junk[80];
38:     gets(junk);
39: }
Enter your age.
29 and never older!
Enter your first name.
Bradley
Your age is 29.
Your name is Bradley.
```

]**ANALYSIS:** When you run Listing 14.7, enter some extra characters after your age, before pressing Enter. Make sure the program ignores them and correctly prompts you for your name. Then modify the program by removing the call to clear_kb(), and run it again. Any extra characters entered on the same line as your age are assigned to name.

Handling Extra Characters with fflush()

There is a second way in which you can clear the extra characters that were typed in. The fflush() function flushes the information in a stream--including the standard input stream. fflush() is generally used with disk files (which are covered on Day 16); however, it can also be used to make Listing 14.7 even simpler. Listing 14.8 uses the fflush() function instead of the clear_kb() function that was created in Listing 14.7.

Listing 14.8. Clearing stdin of extra characters using fflush().

```
1:   /* Clearing stdin of extra characters. */
2:   /* Using the fflush() function         */
3:   #include <stdio.h>
4:
5:   main()
6:   {
7:      int age;
8:      char name[20];
9:
10:     /* Prompt for user's age. */
```

```
11:     puts("Enter your age.");
12:     scanf("%d", &age);
13:
14:     /* Clear stdin of any extra characters. */
15:     fflush(stdin);
16:
17:     /* Now prompt for user's name. */
18:     puts("Enter your first name.");
19:     scanf("%s", name);
20:
21:     /* Display the data. */
22:     printf("Your age is %d.\n", age);
23:     printf("Your name is %s.\n", name);
24:
25:     return 0;
26: }
Enter your age.
29 and never older!
Enter your first name.
Bradley
Your age is 29.
Your name is Bradley.
```

ANALYSIS: As you can see in line 15, the fflush() function is being used. The prototype for the fflush() function is as follows:

```
int fflush( FILE *stream);
```

The *stream* is the stream to be flushed. In Listing 14.8, the standard input stream, stdin, is being passed for *stream*.

scanf() Examples

The best way to become familiar with the operation of the scanf() function is to use it. It's a powerful function, but it can be a bit confusing at times. Try it and see what happens. Listing 14.9 demonstrates some of the unusual ways to use scanf(). You should compile and run this program and then experiment by making changes to the scanf() format strings.

Listing 14.9. Some ways to use scanf() for keyboard input.

```
1:  /* Demonstrates some uses of scanf(). */
2:
3:  #include <stdio.h>
4:
5:
6:
7:  main()
8:  {
9:      int i1, i2;
10:     long l1;
11:
12:     double d1;
13:     char buf1[80], buf2[80];
```

```
14:
15:     /* Using the l modifier to enter long integers and doubles.*/
16:
17:     puts("Enter an integer and a floating point number.");
18:     scanf("%ld %lf", &l1, &d1);
19:     printf("\nYou entered %ld and %lf.\n",l1, d1);
20:     puts("The scanf() format string used the l modifier to
store");
21:     puts("your input in a type long and a type double.\n");
22:
23:     fflush(stdin);
24:
25:     /* Use field width to split input. */
26:
27:     puts("Enter a 5 digit integer (for example, 54321).");
28:     scanf("%2d%3d", &i1, &i2);
29:
30:     printf("\nYou entered %d and %d.\n", i1, i2);
31:     puts("Note how the field width specifier in the scanf()
format");
32:     puts("string split your input into two values.\n");
33:
34:     fflush(stdin);
35:
36:     /* Using an excluded space to split a line of input into */
37:     /* two strings at the space. */
38:
39:     puts("Enter your first and last names separated by a space.");
40:     scanf("%[^ ]%s", buf1, buf2);
41:     printf("\nYour first name is %s\n", buf1);
42:     printf("Your last name is %s\n", buf2);
43:     puts("Note how [^ ] in the scanf() format string, by
excluding");
44:     puts("the space character, caused the input to be split.");
45:
46:     return 0;
47: }
Enter an integer and a floating point number.
123 45.6789
You entered 123 and 45.678900.
The scanf() format string used the l modifier to store
your input in a type long and a type double.
Enter a 5 digit integer (for example, 54321).
54321
You entered 54 and 321.
Note how the field width specifier in the scanf() format
string split your input into two values.
Enter your first and last names separated by a space.
Gayle Johnson
Your first name is Gayle
Your last name is Johnson
```

```
Note how [^ ] in the scanf() format string, by excluding
the space character, caused the input to be split.
```

ANALYSIS: This listing starts by defining several variables in lines 9 through 13 for data input. The program then walks you through the steps of entering various types of data. Lines 17 through 21 have you enter and print long integers and a double. Line 23 calls the fflush() function to clear any unwanted characters from the standard input stream. Lines 27 and 28 get the next value, a five-character integer. Because there are width specifiers, the five-digit integer is split into two integers--one that is two characters, and one that is three characters. Line 34 calls fflush() to clear the keyboard again. The final example, in lines 36 through 44, uses the exclude character. Line 40 uses "%[^]%s", which tells scanf() to get a string but to stop at any spaces. This effectively splits the input.

Take the time to modify this listing and enter additional values to see what the results are.

The scanf() function can be used for most of your input needs, particularly those involving numbers (strings can be input more easily with gets()). It is often worthwhile, however, to write your own specialized input functions. You can see some examples of user-defined functions on Day 18, "Getting More from Functions."

DO take advantage of extended characters in your programs. When using extended characters, you should try to be consistent with other programs.

DON'T forget to check the input stream for extra characters.

DO use the gets() and scanf() functions instead of the fgets() and fscanf() functions if you're using the standard input file (stdin) only.

14-4. Screen Output

Screen output functions are divided into three general categories along the same lines as the input functions: character output, line output, and formatted output. You were introduced to some of these functions in earlier chapters. This section covers them all in detail.

14-4-1. Character Output with putchar(), putc(), and fputc()

The C library's character output functions send a single character to a stream. The function putchar() sends its output to stdout (normally the screen). The functions fputc() and putc() send their output to a stream specified in the argument list.

Using the putchar() Function

The prototype for putchar(), which is located in STDIO.H, is as follows:

```
int putchar(int c);
```

This function writes the character stored in c to stdout. Although the prototype specifies a type int argument, you pass putchar() a type char. You can also pass it a type int as long as its value is appropriate for a character (that is, in the range 0 to 255). The function returns the character that was just written, or EOF if an error has occurred.

You saw putchar() demonstrated in Listing 14.2. Listing 14.10 displays the characters with ASCII values between 14 and 127.

Listing 14.10. The putchar() function.

```
1: /* Demonstrates putchar(). */
2:
3: #include <stdio.h>
4: main()
5: {
6:     int count;
7:
8:     for (count = 14; count < 128; )
9:          putchar(count++);
10:
11:   return 0;
12: }
```

You can also display strings with the putchar() function (as shown in Listing 14.11), although other functions are better suited for this purpose.

Listing 14.11. Displaying a string with putchar().

```
1: /* Using putchar() to display strings. */
2:
3: #include <stdio.h>
4:
5: #define MAXSTRING 80
6:
7: char message[] = "Displayed with putchar().";
8: main()
9: {
10:     int count;
11:
12:     for (count = 0; count < MAXSTRING; count++)
13:     {
14:
15:         /* Look for the end of the string. When it's found, */
16:         /* write a newline character and exit the loop. */
17:
18:         if (message[count] == `\0')
19:         {
20:             putchar(`\n');
21:             break;
22:         }
23:         else
24:
25:         /* If end of string not found, write the next character.
*/
26:
27:             putchar(message[count]);
28:     }
29:     return 0;
30: }
Displayed with putchar().
```

Using the putc() and fputc() Functions

These two functions perform the same action--sending a single character to a specified stream. putc() is a macro implementation of fputc(). You'll learn about macros on Day 21, "Advanced Compiler Use." For now, just stick to fputc(). Its prototype is

```
int fputc(int c, FILE *fp);
```

The FILE *fp part might puzzle you. You pass fputc() the output stream in this argument. (You'll learn more about this on Day 16.) If you specify stdout as the stream, fputc() behaves exactly the same as putchar(). Thus, the following two statements are equivalent:

```
putchar(`x');
fputc(`x', stdout);
```

14-4-2. Using puts() and fputs() for String Output

Your programs display strings on-screen more often than they display single characters. The library function puts() displays strings. The function fputs() sends a string to a specified stream; otherwise, it is identical to puts(). The prototype for puts() is

```
int puts(char *cp);
```

*cp is a pointer to the first character of the string that you want displayed. The puts() function displays the entire string up to but not including the terminating null character, adding a newline at the end. Then puts() returns a positive value if successful or EOF on error. (Remember, EOF is a symbolic constant with the value -1; it is defined in STDIO.H.)

The puts() function can be used to display any type of string, as demonstrated in Listing 14.12.

Listing 14.12. Using the puts() function to display strings.

```
1: /* Demonstrates puts(). */
2:
3: #include <stdio.h>
4:
5: /* Declare and initialize an array of pointers. */
6:
7: char *messages[5] = { "This", "is", "a", "short", "message." };
8:
9: main()
10: {
11:     int x;
12:
13:     for (x=0; x<5; x++)
14:         puts(messages[x]);
15:
16:     puts("And this is the end!");
17:
18:     return 0;
19: }
This
is
a
short
```

```
message.
And this is the end!
```

ANALYSIS: This listing declares an array of pointers, a subject not covered yet. (It will be covered tomorrow.) Lines 13 and 14 print each of the strings stored in the message array.

14-4-3. Using printf() and fprintf() for Formatted Output

So far, the output functions have displayed characters and strings only. What about numbers? To display numbers, you must use the C library's formatted output functions, printf() and fprintf(). These functions can also display strings and characters. You were officially introduced to printf() on Day 7, and you've used it in almost every chapter. This section provides the remainder of the details.

The two functions printf() and fprintf() are identical, except that printf() always sends output to stdout, whereas fprintf() specifies the output stream. fprintf() is generally used for output to disk files. It's covered on Day 16.

The printf() function takes a variable number of arguments, with a minimum of one. The first and only required argument is the format string, which tells printf() how to format the output. The optional arguments are variables and expressions whose values you want to display. Take a look at these few simple examples, which give you a feel for printf(), before you really get into the nitty-gritty:

- The statement printf("Hello, world."); displays the message Hello, world. on-screen. This is an example of using printf() with only one argument, the format string. In this case, the format string contains only a literal string to be displayed on-screen.
- The statement printf("%d", i); displays the value of the integer variable i on-screen. The format string contains only the format specifier %d, which tells printf() to display a single decimal integer. The second argument i is the name of the variable whose value is to be displayed.
- The statement printf("%d plus %d equals %d.", a, b, a+b); displays 2 plus 3 equals 5 on-screen (assuming that a and b are integer variables with the values of 2 and 3, respectively). This use of printf() has four arguments: a format string that contains literal text as well as format specifiers, and two variables and an expression whose values are to be displayed.

Now look at the printf() format string in more detail. It can contain the following:

- Zero, one, or more conversion commands that tell printf() how to display a value in its argument list. A conversion command consists of % followed by one or more characters.
- Characters that are not part of a conversion command and are displayed as-is.

The third example's format string is %d plus %d equals %d. In this case, the three %ds are conversion commands, and the remainder of the string, including the spaces, is literal characters that are displayed directly.

Now you can dissect the conversion command. The components of the command are given here and explained next. Components in brackets are optional.

%[*flag*][*field_width*][.[*precision*]][l]*conversion_char*

The *conversion_char* is the only required part of a conversion command (other than the %). Table 14.5 lists the conversion characters and their meanings.

Table 14.5. The printf() and fprintf() conversion characters.

Conversion Character	Meaning
d, I	Display a signed integer in decimal notation.

U	Display an unsigned integer in decimal notation.
O	Display an integer in unsigned octal notation.
x, X	Display an integer in unsigned hexadecimal notation. Use x for lowercase output and X for uppercase output.
C	Display a single character (the argument gives the character's ASCII code).
e, E	Display a float or double in scientific notation (for example, 123.45 is displayed as 1.234500e+002). Six digits are displayed to the right of the decimal point unless another precision is specified with the f specifier. Use e or E to control the case of output.
F	Display a float or double in decimal notation (for example, 123.45 is displayed as 123.450000). Six digits are displayed to the right of the decimal point unless another precision is specified.
g, G	Use e, E, or f format. The e or E format is used if the exponent is less than -3 or greater than the precision (which defaults to 6). f format is used otherwise. Trailing zeros are truncated.
N	Nothing is displayed. The argument corresponding to an n conversion command is a pointer to type int. The printf() function assigns to this variable the number of characters output so far.
S	Display a string. The argument is a pointer to char. Characters are displayed until a null character is encountered or the number of characters specified by precision (which defaults to 32767) is displayed. The terminating null character is not output.
%	Display the % character.

You can place the l modifier just before the conversion character. This modifier applies only to the conversion characters o, u, x, X, i, d, and b. When applied, this modifier specifies that the argument is a type long rather than a type int. If the l modifier is applied to the conversion characters e, E, f, g, or G, it specifies that the argument is a type double. If an l is placed before any other conversion character, it is ignored.

The precision specifier consists of a decimal point (.) by itself or followed by a number. A precision specifier applies only to the conversion characters e, E, f, g, G, and s. It specifies the number of digits to display to the right of the decimal point or, when used with s, the number of characters to output. If the decimal point is used alone, it specifies a precision of 0.

The field-width specifier determines the minimum number of characters output. The field-width specifier can be the following:

- A decimal integer not starting with 0. The output is padded on the left with spaces to fill the designated field width.
- A decimal integer starting with 0. The output is padded on the left with zeros to fill the designated field width.
- The * character. The value of the next argument (which must be an int) is used as the field width. For example, if w is a type int with a value of 10, the statement printf("%*d", w, a); prints the value of a with a field width of 10.

If no field width is specified, or if the specified field width is narrower than the output, the output field is just as wide as needed.

The last optional part of the printf() format string is the flag, which immediately follows the % character. There are four available flags:

- This means that the output is left-justified in its field rather than right-justified, which is the default.

+ This means that signed numbers are always displayed with a leading + or -.

` ` A space means that positive numbers are preceded by a space.

This applies only to x, X, and o conversion characters. It specifies that nonzero numbers are displayed with a leading 0X or 0x (for x and X) or a leading 0 (for o).

When you use printf(), the format string can be a string literal enclosed in double quotes in the printf() argument list. It can also be a null-terminated string stored in memory, in which case you pass a pointer to the string to printf(). For example, this statement:

```
char *fmt = "The answer is %f.";
printf(fmt, x);
```

is equivalent to this statement:

```
printf("The answer is %f.", x);
```

As explained on Day 7, the printf() format string can contain escape sequences that provide special control over the output. Table 14.6 lists the most frequently used escape sequences. For example, including the newline sequence (\n) in a format string causes subsequent output to appear starting on the next screen line.

Table 14.6. The most frequently used escape sequences.

Sequence	Meaning
\a	Bell (alert)
\b	Backspace
\n	Newline
\t	Horizontal tab
\\	Backslash
\?	Question mark
\'	Single quote
\"	Double quote

printf() is somewhat complicated. The best way to learn how to use it is to look at examples and then experiment on your own. Listing 14.13 demonstrates some of the ways you can use printf().

Listing 14.13. Some ways to use the printf() function.

```
1:  /* Demonstration of printf(). */
2:
3:  #include <stdio.h>
4:
5:  char *m1 = "Binary";
6:  char *m2 = "Decimal";
7:  char *m3 = "Octal";
8:  char *m4 = "Hexadecimal";
9:
```

```
10: main()
11: {
12:     float d1 = 10000.123;
13:     int n, f;
14:
15:
16:     puts("Outputting a number with different field widths.\n");
17:
18:     printf("%5f\n", d1);
19:     printf("%10f\n", d1);
20:     printf("%15f\n", d1);
21:     printf("%20f\n", d1);
22:     printf("%25f\n", d1);
23:
24:     puts("\n Press Enter to continue...");
25:     fflush(stdin);
26:     getchar();
27:
28:     puts("\nUse the * field width specifier to obtain field
width");
29:     puts("from a variable in the argument list.\n");
30:
31:     for (n=5;n<=25; n+=5)
32:         printf("%*f\n", n, d1);
33:
34:     puts("\n Press Enter to continue...");
35:     fflush(stdin);
36:     getchar();
37:
38:     puts("\nInclude leading zeros.\n");
39:
40:     printf("%05f\n", d1);
41:     printf("%010f\n", d1);
42:     printf("%015f\n", d1);
43:     printf("%020f\n", d1);
44:     printf("%025f\n", d1);
45:
46:     puts("\n Press Enter to continue...");
47:     fflush(stdin);
48:     getchar();
49:
50:     puts("\nDisplay in octal, decimal, and hexadecimal.");
51:     puts("Use # to precede octal and hex output with 0 and 0X.");
52:     puts("Use - to left-justify each value in its field.");
53:     puts("First display column labels.\n");
54:
55:     printf("%-15s%-15s%-15s", m2, m3, m4);
56:
57:     for (n = 1;n< 20; n++)
58:         printf("\n%-15d%-#15o%-#15X", n, n, n);
59:
```

```
60:     puts("\n Press Enter to continue...");
61:     fflush(stdin);
62:     getchar();
63:
64:     puts("\n\nUse the %n conversion command to count
characters.\n");
65:
66:     printf("%s%s%s%s%n", m1, m2, m3, m4, &n);
67:
68:     printf("\n\nThe last printf() output %d characters.\n", n);
69:
70:     return 0;
71: }
Outputting a number with different field widths.
10000.123047
10000.123047
   10000.123047
        10000.123047
             10000.123047
 Press Enter to continue...
Use the * field width specifier to obtain field width
from a variable in the argument list.
10000.123047
10000.123047
   10000.123047
        10000.123047
             10000.123047
 Press Enter to continue...
Include leading zeros.
10000.123047
10000.123047
00010000.123047
0000000010000.123047
000000000000010000.123047
 Press Enter to continue...
Display in octal, decimal, and hexadecimal.
Use # to precede octal and hex output with 0 and 0X.
Use - to left-justify each value in its field.
First display column labels.
Decimal        Octal          Hexadecimal
1              01             0X1
2              02             0X2
3              03             0X3
4              04             0X4
5              05             0X5
6              06             0X6
7              07             0X7
8              010            0X8
9              011            0X9
10             012            0XA
11             013            0XB
```

```
12              014             0XC
13              015             0XD
14              016             0XE
15              017             0XF
16              020             0X10
17              021             0X11
18              022             0X12
19              023             0X13
 Press Enter to continue...
Use the %n conversion command to count characters.
BinaryDecimalOctalHexadecimal
The last printf() output 29 characters.
```

14-5. Redirecting Input and Output

A program that uses stdin and stdout can utilize an operating-system feature called *redirection*. Redirection allows you to do the following:

- Output sent to stdout can be sent to a disk file or the printer rather than to the screen.
- Program input from stdin can come from a disk file rather than from the keyboard.

You don't code redirection into your programs; you specify it on the command line when you run the program. In DOS, as in UNIX, the symbols for redirection are < and >. I'll discuss redirection of output first.

Remember your first C program, HELLO.C? It used the printf() library function to display the message Hello, world on-screen. As you now know, printf() sends output to stdout, so it can be redirected. When you enter the program name at the command-line prompt, follow it with the > symbol and the name of the new destination:

```
hello > destination
```

Thus, if you enter hello >prn, the program output goes to the printer instead of to the screen (prn is the DOS name for the printer attached to port LPT1:). If you enter hello >hello.txt, the output is placed in a disk file with the name HELLO.TXT.

When you redirect output to a disk file, be careful. If the file already exists, the old copy is deleted and replaced with the new file. If the file doesn't exist, it is created. When redirecting output to a file, you can also use the >> symbol. If the specified destination file already exists, the program output is appended to the end of the file. Listing 14.14 demonstrates redirection.

Listing 14.14. The redirection of input and output.

```
1: /* Can be used to demonstrate redirection of stdin and stdout. */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7:    char buf[80];
8:
9:    gets(buf);
10:   printf("The input was: %s\n", buf);
11:   return 0;
12: }
```

ANALYSIS: This program accepts a line of input from stdin and then sends the line to stdout, preceding it with The input was:. After compiling and linking the program, run it without redirection (assuming that the program is named LIST1414) by entering LIST1414 at the command-line prompt. If you then enter I am teaching myself C, the program displays the following on-screen:

```
The input was: I am teaching myself C
```

If you run the program by entering LIST1414 >test.txt and make the same entry, nothing is displayed on-screen. Instead, a file named TEST.TXT is created on the disk. If you use the DOS TYPE (or an equivalent) command to display the contents of the file:

```
type test.txt
```

you'll see that the file contains only the line The input was: I am teaching myself C. Similarly, if you had run the program by entering LIST1414 >prn, the output line would have been printed on the printer (prn is a DOS command name for the printer).

Run the program again, this time redirecting output to TEST.TXT with the >> symbol. Instead of the file's getting replaced, the new output is appended to the end of TEST.TXT.

14-5-1. Redirecting Input

Now let's look at redirecting input. First you need a source file. Use your editor to create a file named INPUT.TXT that contains the single line William Shakespeare. Now run Listing 14.14 by entering the following at the DOS prompt:

```
list1414 < INPUT.TXT
```

The program doesn't wait for you to make an entry at the keyboard. Instead, it immediately displays the following message on-screen:

```
The input was: William Shakespeare
```

The stream stdin was redirected to the disk file INPUT.TXT, so the program's call to gets() reads one line of text from the file rather than the keyboard.

You can redirect input and output at the same time. Try running the program with the following command to redirect stdin to the file INPUT.TXT and redirect stdout to JUNK.TXT:

```
list1414 < INPUT.TXT > JUNK.TXT
```

Redirecting stdin and stdout can be useful in certain situations. A sorting program, for example, could sort either keyboard input or the contents of a disk file. Likewise, a mailing list program could display addresses on-screen, send them to the printer for mailing labels, or place them in a file for some other use.

NOTE: Remember that redirecting stdin and stdout is a feature of the operating system and not of the C language itself. However, it does provide another example of the flexibility of streams. You can check your operating system documentation for more information on redirection.

14-6. When to Use fprintf()

As mentioned earlier, the library function fprintf() is identical to printf(), except that you can specify the stream to which output is sent. The main use of fprintf() involves disk files, as explained on Day 16. There are two other uses, as explained here.

14-6-1. Using stderr

One of C's predefined streams is stderr (standard error). A program's error messages traditionally are sent to the stream stderr and not to stdout. Why is this?

As you just learned, output to stdout can be redirected to a destination other than the display screen. If stdout is redirected, the user might not be aware of any error messages the program sends to stdout. Unlike stdout, stderr can't be redirected and is always connected to the screen (at least in DOS--UNIX systems might allow redirection of stderr). By directing error messages to stderr, you can be sure the user always sees them. You do this with fprintf():

```
fprintf(stderr, "An error has occurred.");
```

You can write a function to handle error messages and then call the function when an error occurs rather than calling fprintf():

```
error_message("An error has occurred.");
void error_message(char *msg)
{
    fprintf(stderr, msg);
}
```

By using your own function instead of directly calling fprintf(), you provide additional flexibility (one of the advantages of structured programming). For example, in special circumstances you might want a program's error messages to go to the printer or a disk file. All you need to do is modify the error_message() function so that the output is sent to the desired destination.

Printer Output Under DOS

On a DOS or Windows system, you send output to your printer by accessing the predefined stream stdprn. On IBM PCs and compatibles, the stream stdprn is connected to the device LPT1: (the first parallel printer port). Listing 14.15 presents a simple example.

NOTE: To use stdprn, you need to turn ANSI compatibility off in your compiler. Consult your compiler's manuals for more information.

Listing 14.15. Sending output to the printer.

```
1: /* Demonstrates printer output. */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7:     float f = 2.0134;
8:
9:      fprintf(stdprn, "\nThis message is printed.\r\n");
```

```
10:     fprintf(stdprn, "And now some numbers:\r\n");
11:     fprintf(stdprn, "The square of %f is %f.", f, f*f);
12:
13:     /* Send a form feed. */
14:     fprintf(stdprn, "\f");
15:
16:     return 0;
17: }
This message is printed.
And now some numbers:
The square of 2.013400 is 4.053780.
```

NOTE: This output is printed by the printer. It won't appear on-screen.

ANALYSIS: If your DOS system has a printer connected to port LPT1:, you can compile and run this program. It prints three lines on the page. Line 14 sends an "\f" to the printer. \f is the escape sequence for a form feed, the command that causes the printer to advance a page (or, in the case of a laser printer, to eject the current page).

DON'T ever try to redirect stderr.

DO use fprintf() to create programs that can send output to stdout, stderr, stdprn, or any other stream.

DO use fprintf() with stderr to print error messages to the screen.

DON'T use stderr for purposes other than printing error messages or warnings.

DO create functions such as error_message to make your code more structured and maintainable.

14-7. Summary

This was a long day full of important information on program input and output. You learned how C uses streams, treating all input and output as a sequence of bytes. You also learned that C has five predefined streams:

Stdin	The keyboard
Stdout	The screen
Stderr	The screen
Stdprn	The printer
Stdaux	The communications port

Input from the keyboard arrives from the stream stdin. Using C's standard library functions, you can accept keyboard input character by character, a line at a time, or as formatted numbers and strings. Character input can be buffered or unbuffered, echoed or unechoed.

Output to the display screen is normally done with the stdout stream. Like input, program output can be by character, by line, or as formatted numbers and strings. For output to the printer, you use fprintf() to send data to the stream stdprn.

When you use stdin and stdout, you can redirect program input and output. Input can come from a disk file rather than the keyboard, and output can go to a disk file or to the printer rather than to the display screen.

Finally, you learned why error messages should be sent to the stream stderr instead of stdout. Because stderr is usually connected to the display screen, you are assured of seeing error messages even when the program output is redirected.

Q&A

Q What happens if I try to get input from an output stream?
A You can write a C program to do this, but it won't work. For example, if you try to use stdprn with fscanf(), the program compiles into an executable file, but the printer is incapable of sending input, so your program doesn't operate as intended.
Q What happens if I redirect one of the standard streams?
A Doing this might cause problems later in the program. If you redirect a stream, you must put it back if you need it again in the same program. Many of the functions described in this chapter use the standard streams. They all use the same streams, so if you change the stream in one place, you change it for all the functions. For example, assign stdout equal to stdprn in one of the listings in this chapter and see what happens.
Q Is there any danger in using non-ANSI functions in a program?
A Most compilers come with many useful functions that aren't ANSI-standard. If you plan on always using that compiler and not porting your code to other compilers or platforms, there won't be a problem. If you're going to use other compilers and platforms, you should be concerned with ANSI compatibility.
Q Why shouldn't I always use fprintf() instead of printf()? Or fscanf() instead of scanf()?
A If you're using the standard output or input streams, you should use printf() and scanf(). By using these simpler functions, you don't have to bother with any other streams.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.

Quiz

1. What is a stream, and what does a C program use streams for?
2. Are the following input devices or output devices?
a. Printer
b. Keyboard
c. Modem
d. Monitor
e. Disk drive
3. List the five predefined streams and the devices with which they are associated.
4. What stream do the following functions use?
a. printf()
b. puts()
c. scanf()
d. gets()
e. fprintf()
5. What is the difference between buffered and unbuffered character input from stdin?
6. What is the difference between echoed and unechoed character input from stdin?
7. Can you "unget" more than one character at a time with ungetc()? Can you "unget" the EOF character?
8. When you use C's line input functions, how is the end of a line determined?
9. Which of the following are valid type specifiers?
a. "%d"

b. "%4d"
c. "%3i%c"
d. "%q%d"
e. "%%%i"
f. "%9ld"
10. What is the difference between stderr and stdout?

Exercises

1. Write a statement to print "Hello World" to the screen.
2. Use two different C functions to do the same thing the function in exercise 1 did.
3. Write a statement to print "Hello Auxiliary Port" to the standard auxiliary port.
4. Write a statement that gets a string 30 characters or shorter. If an asterisk is encountered, truncate the string.
5. Write a single statement that prints the following:

```
Jack asked, "What is a backslash?"
Jill said, "It is `\'"
```

Because of the multitude of possibilities, answers are not provided for the following exercises; however, you should attempt to do them.
6. ON YOUR OWN: Write a program that redirects a file to the printer one character at a time.
7. ON YOUR OWN: Write a program that uses redirection to accept input from a disk file, counts the number of times each letter occurs in the file, and then displays the results on-screen. (A hint is provided in Appendix G, "Answers.")
8. ON YOUR OWN: Write a program that prints your C source files. Use redirection to enter the source file, and use fprintf() to do the printing.
9. ON YOUR OWN: Modify the program from exercise 8 to put line numbers at the beginning of the listing when it is printed. (A hint is provided in Appendix G.)
10. ON YOUR OWN: Write a "typing" program that accepts keyboard input, echoes it to the screen, and then reproduces this input on the printer. The program should count lines and advance the paper in the printer to a new page when necessary. Use a function key to terminate the program.